DOCUMENTATION FOR THE TOKEN RING NETWORK SIMULATION SYSTEM

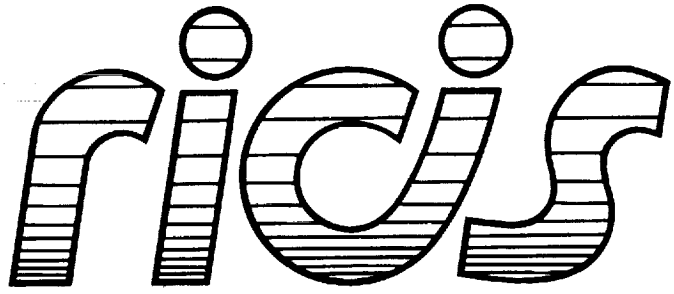
Jeffery H. Peden
Alfred C. Weaver

Digital Technology

November 1990

Cooperative Agreement NCC 9-16
Research Activity No. SE.31

NASA Johnson Space Center
Engineering Directorate
Flight Data Systems Division



Research Institute for Computing and Information Systems
University of Houston - Clear Lake

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information systems in 1986 to encourage NASA Johnson Space Center and local industry to actively support research in the computing and information sciences. As part of this endeavor, UH-Clear Lake proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a three-year cooperative agreement with UH-Clear Lake beginning in May, 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The mission of RICIS is to conduct, coordinate and disseminate research on computing and information systems among researchers, sponsors and users from UH-Clear Lake, NASA/JSC, and other research organizations. Within UH-Clear Lake, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business, Education, Human Sciences and Humanities, and Natural and Applied Sciences.

Other research organizations are involved via the "gateway" concept. UH-Clear Lake establishes relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research.

A major role of RICIS is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. Working jointly with NASA/JSC, RICIS advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research, and integrates technical results into the cooperative goals of UH-Clear Lake and NASA/JSC.

Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Jeffery H. Peden, Alfred C. Weaver and Digital Technology. Dr. George Collins, Associate Professor of Computer Systems Design, served as RICIS technical representative for this activity.

Funding has been provided by the Engineering Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Frank W. Miller, of the Systems Development Branch, Flight Data Systems Division, Engineering Directorate, NASA/JSC.

The views and conclusions contained in this report are those of the authors and should not be interpreted as representative of the official policies, either express or implied, of NASA or the United States Government.

Documentation
for the
Token Ring Network Simulation System

Jeffery H. Peden
Alfred C. Weaver
Digital Technology
Charlottesville, Virginia

Introduction

This manual describes the language features of the Token Ring Network Simulation System. The simulation system is a powerful simulation tool for token ring networks which allows the specification of various MAC layer protocols as well as the specification of various features of upper layer ISO protocols. In addition to these features, it also allows the user to specify message and station classes virtually to any degree of detail desired.

The choice of using a language instead of an interactive system to specify network parameters was dictated by both flexibility and time considerations. Certainly an interactive system is immediately usable whereas a language based system requires prior study, but after the language is learned (only about an hour of study is required for this particular language), it takes less time to use than an interactive system. Also, it is possible to be more detailed with less effort if a language is used rather than an interactive system.

The language presented here was developed specifically for the simulation system, and is very simple. It is also user friendly in that language elements which do not apply to the case at hand are ignored rather than treated as errors. A third feature of this language is its self documenting feature — a glance at the written language allows the user to immediately see what network configuration is being programmed.

Purpose of the Simulation System

The purpose of any simulation system is to enable the network designer and operator to predict certain operating characteristics of a network without actually using the network; the main purpose of this simulation system is no different. What is different, however, is the scope of network activity covered by the simulator.

This simulation system is intended not only to simulate the MAC layer of a network, but also to simulate the message processing layers which feed into and take messages from the MAC layer. This allows the network designer to gain a more accurate picture of the actual end-to-end delay encountered by a typical message.

The system is also intended to simulate a more accurate representation of message traffic than most other systems. This system allows the user to set up various message classes and priorities, thus gaining a better view of the actual operational characteristics of the network.

Structure of the Simulation System

The structure of the simulation system is modular in nature. A token ring network has two distinct operational characteristics: the MAC layer operation, which is dependent on the token cycle; and the upper layer operation, which is not. The simulation system's structure and operational characteristics reflect this dual nature of a ring network. There is a module for MAC layer operation, and one for the upper layer operation (each upper layer is sufficiently similar in operation that it is not necessary for each layer to be a separate module).

The coupling of procedures within modules and between modules is very loose since there are no global variables — information is passed between procedures and functions only via parameters. The only global information consists of the various type definitions. The modules themselves exhibit a mix of functional and temporal cohesion.

Internal Operation of the Simulation System

The simulation system is a *continuous time discrete event* simulator. The events which drive the simulation are the token arrivals at the network stations, and message transmissions (note that the movement of a message from one network layer to another is considered to be a message transmission). The internal, or logical time of the simulator is a floating point variable which is in units of bit times, and is updated whenever an event occurs.

Each token arrival event triggers various internal simulator events, such as message transmission, message enqueueing, delay and queue length measurements, upper layer processing, and logical-time clock updating. Messages are enqueued and dequeued by adding and deleting nodes from linked lists. The network itself is represented as multi-dimensional array, one index of which is considered to be the network stations. Each cell of the array contains pointers to various linked lists, such as the linked lists of messages enqueued at each message processing layer. The circulation of the token is caused by circular indexing of the array of network stations.

Example Program

The following is an example input file for use with the token ring simulator.

```
# input file for simulator program
# general
# this configuration is simple -- only a single class of messages, wit
# every station participating
# all messages enter the network at the MAC layer
FDDI
iterations 6
stations 10
circumference 100.0
bps 100000000
latency 32
simulation 0.02
rho 95.0
threshold 0 9.5
#
```

```
# class related stuff
#
class 0
#
includes all
enter mac
load 100.0
length 32
priority 0 100.0
messages deterministic
arrivals poisson
```

Language Description

This language allows the user to set up a simulation with detailed information concerning the number of ring stations, classes of messages, arrival and service distributions for classes, etc. Classes and stations are entirely orthogonal -- there may be multiple stations per message class, and multiple message classes per station. This feature allows the maximum flexibility when setting up a simulation.

All input between a # and a newline character is ignored, thus allowing for comments.

General Network Commands

There are several network-wide specification commands. These are

```
FDDI
iterations N
stations N
circumference X
meters N X
bps N
latency N
simulation X
rho X
threshold N X
tokenhold X
preemption
```

The symbol `N` indicates an integer value, and the symbol `X` indicates a floating-point value (floating point values are not required, however).

The notation `FDDI` allows the user to specify which mac layer protocol is to be used (currently only the FDDI network may be selected).

`iterations N` specifies the number of independent replications of the simulation that are to be done. This allows whatever statistical significance the user desires.

`stations N` specifies the number of stations on the network.

`circumference X` sets the total ring circumference, where `X` is expected to be in meters. A floating point value is allowed. The distance between any two stations is found by dividing the circumference by the number of stations, subject to the `meters N X` command.

`meters N X` allows the specification of the distance between any two stations on the ring, with `XR in meters.` The command `meters 34 4.5` has the effect that the distance from ring station 34 to ring station 35 is 4.5 meters. This overrides the effect of the `circumference` command. A further effect of this command is that the distances between all other stations are automatically adjusted so that the total circumference remains as specified in the `circumference` command. The `meters` command may be repeated as many times as

is necessary. It may also be used exclusively without the `circumference` command.

`bps N` sets the data rate of the ring in bits per second. The MAC layer protocols in the simulation system do not assume a default value (even if the protocol being simulated requires a certain value), so this command must be used.

`latency N` controls the value of the internal station latency. The value is in bit times. For the FDDI protocol, this value varies with the manufacturer of the chipset.

`simulation X` sets the number of seconds of simulation time (with `X` in seconds), e.g., 3.0 seconds, 0.02 seconds, etc. Note that this is not real time, but rather internal simulator time. The simulator typically takes much longer than 0.2 seconds, for example, to simulate 0.2 seconds of network operation. (This is due to the fact that the simulation is done on a uniprocessor, whereas networks inherently have many processors implementing their own local protocol functions.)

`rho X` sets the value of the overall network load as a percentage. A floating point value is expected, and may be greater than 100.0 if overload conditions are to be simulated.

The command `threshold N X` is MAC layer protocol specific. This command sets the priority threshold for priority `N` at `X` milliseconds. This only has an effect if the FDDI protocol has been selected.

The `preemption` command controls the priority operation in the upper layers. If present, higher priority messages preempt the processor from lower priority messages. If absent, messages are processed in their entirety, and then the highest priority waiting message is processed.

Message Class Specific Commands

The class information commands allow different classes of messages to be configured. Any number of classes may be specified. There may be many stations having message class K, and there may be many message classes at any particular station. Note that we are defining *message* classes, not station classes. This mechanism is sufficiently general, however, that station classes may be defined using this method merely by having any particular station only offer a single message class to the network.

The class commands are

```
class N
includes [all,even,odd,N1 [N2 ...],[N3-N4]]
enter [mac,llc,network,transport,session,presentation,application]
load X
length N
priority N X
messages [uniform,constant,exponential]
arrivals [poisson,deterministic,uniform]
```

`class N` begins the definition for message class `N`, where `N` can be any arbitrary number (classes do not have to be numbered in order, and class numbers may be left out, so long as there is at least one class). The other class commands which follow apply to message class `N` until another `class N` command is encountered.

Stations which handle message class `N` are specified by the `includes` command. The command `includes all` specifies that all network stations offer this class. Similarly, `includes even` and `includes odd` specify that only even or odd stations respectively offer this class. It is possible to specify arbitrary stations by `includes 2 4 7 9 32`, for example, which says that stations 2, 4, 7, 9, and 32 offer message class `N` (any number of stations may be specified in this manner). The notation `includes [2-14]` indicates that stations 2 through 14 offer this message class.

The above station specification commands may be combined, with the result that the union of the specifications is the group of included stations. That is, the command `includes even 3 5 41` results in all even numbered stations offering message class `N`, as well as the additional stations 3, 5, and 41. The command `includes even odd` is identical to the command `includes all`. The command `includes odd [2-10]` results in all odd stations being included as well as stations 2, 4, 6, 8, and 10 (stations 3, 5, 7, and 9 were already included).

`enter` indicates at which level of the protocol stack messages of the currently defined class enter (and leave).

`load X` specifies the offered load, expressed as the *percentage* of `rho` (see above) this message class contributes.

`length N` specifies the mean length of messages of this class in *bytes*.

`priority N X` sets the *percentage* of `load` (see above) that priority `N` messages of this class are to receive. Note that not all priority levels need be represented in any particular message class.

`messages` indicates the message length distribution. `arrivals` indicates the arrival process distribution. Other distributions will probably be added in the near future. Currently the simulator will not handle a trace (this feature can be added if it is deemed necessary or desirable).

Upper Layer Specific Commands

The upper layer commands are

```
application
presentation
session
transport
network
llc
```

Note that if these commands are preceded by the command `enter` they are not upper layer commands. All upper layer commands must be included that are equal to or below the highest entry layer of any message class.

Each upper layer command *must be followed by*

```
send X X
rec X X
framing N
```

and may also optionally be followed by

```
segment N
```

Each of `send X X`, `rec X X`, `framing N`, and `segment N` all control these parameters for the immediately preceding layer specification.

`send X X` and `rec X X` indicate (a) the call time overhead (essentially the time needed for the target processor to implement a procedure call), and (b) the processing time per *byte* for that layer (this is independent of message class and priority since this is a hardware/software issue). The command `send 5.0 200.0` means that the call time overhead is 5.0 milliseconds, and the processing time per byte is 200.0 nanoseconds. `rec 3.0 500.0` has a similar interpretation. The `framing N` command results in N bytes of framing being added to the message. Note that message length has already be specified using the class commands; the result is that all framing added in upper layers fills out the message to be the final specified length. The `segment N` command, if present, specifies that messages of length greater than N

are to be segmented at that layer. If absent, no segmentation takes place.

Simulator Output

Simulator output is sent to the screen unless it is redirected using the notation `simulate <filename>`. As the simulator runs, its progress (relative to the length of the simulation) is sent to the screen regardless of whether or not the simulator output is to be sent to the screen. The command `simulate results1` causes the simulator output to be written to the file "results1".

The user need not be concerned with overwriting an existing file by accident, because if the file used in the `simulate <filename>` already exists, the new results are appended to its end. The date and time of the simulation are prepended to the results in all cases. If the user wishes the output sent to the screen, the command `simulate` is used. This is, however, unwise (unless, for example, a log file is being created) since the output in all cases takes up more than 24 lines (a typical screen length).

The following is an example output file.

```
*****************************
*****************************

Sun Aug 20 17:17:13 EDT 1995

entry layer offered load = 0.960
mac level offered load = 0.497
mac throughput = 0.497
end throughput = 0.016
token cycle time = 1.524e-05
mac queue length = 1.380e+00
LLC layer queue length:
   transmit = 1.154e+00
    receive = 0.000e+00
NETWORK layer queue length:
   transmit = 0.000e+00
    receive = 0.000e+00
TRANSPORT layer queue length:
   transmit = 9.323e+01
    receive = 0.000e+00
TRANSPORT layer queueing delay:
```

```
   transmit = 3.332e-03
    receive = 0.000e+00
NETWORK layer queueing delay:
   transmit = 0.000e+00
    receive = 0.000e+00
LLC layer queueing delay:
   transmit = 6.531e-03
    receive = 0.000e+00
queueing delay = 5.248e-05
net access delay = 3.812e-05
mean wait delay = 9.059e-05
priority 6
   token cycle time = 1.524e-05
   mac queue length = 6.864e-02
   LLC layer queue length:
      transmit = 4.515e-01
       receive = 0.000e+00
   NETWORK layer queue length:
      transmit = 0.000e+00
       receive = 0.000e+00
   TRANSPORT layer queue length:
      transmit = 3.781e+01
       receive = 0.000e+00
   TRANSPORT layer queueing delay:
      transmit = 3.609e-03
       receive = 0.000e+00
   NETWORK layer queueing delay:
      transmit = 0.000e+00
       receive = 0.000e+00
   LLC layer queueing delay:
      transmit = 6.808e-03
       receive = 0.000e+00
   queueing delay = 2.424e-03
   net access delay = 2.196e-03
   mean wait delay = 4.620e-03
      class 1
         mac queue length = 6.864e-02
         LLC layer queue length:
            transmit = 4.515e-01
             receive = 0.000e+00
         NETWORK layer queue length:
            transmit = 0.000e+00
             receive = 0.000e+00
         TRANSPORT layer queue length:
            transmit = 3.781e+01
             receive = 0.000e+00
         TRANSPORT layer queueing delay:
            transmit = 3.305e-03
             receive = 0.000e+00
         NETWORK layer queueing delay:
            transmit = 0.000e+00
```

```
            receive = 0.000e+00
         LLC layer queueing delay:
            transmit = 6.504e-03
             receive = 0.000e+00
         queueing delay = 2.424e-03
         net access delay = 2.196e-03
         mean wait delay = 4.620e-03

priority 7
   token cycle time = 1.524e-05
   mac queue length = 7.599e-02
   LLC layer queue length:
      transmit = 6.629e-01
       receive = 0.000e+00
   NETWORK layer queue length:
      transmit = 0.000e+00
       receive = 0.000e+00
   TRANSPORT layer queue length:
      transmit = 5.442e+01
       receive = 0.000e+00
   TRANSPORT layer queueing delay:
      transmit = 3.532e-03
       receive = 0.000e+00
   NETWORK layer queueing delay:
      transmit = 0.000e+00
       receive = 0.000e+00
   LLC layer queueing delay:
      transmit = 6.731e-03
       receive = 0.000e+00
   queueing delay = 3.144e-03
   net access delay = 2.300e-03
   mean wait delay = 5.444e-03
      class 1
         mac queue length = 7.599e-02
         LLC layer queue length:
            transmit = 6.629e-01
             receive = 0.000e+00
         NETWORK layer queue length:
            transmit = 0.000e+00
             receive = 0.000e+00
         TRANSPORT layer queue length:
            transmit = 5.442e+01
             receive = 0.000e+00
         TRANSPORT layer queueing delay:
            transmit = 3.108e-03
             receive = 0.000e+00
         NETWORK layer queueing delay:
            transmit = 0.000e+00
             receive = 0.000e+00
         LLC layer queueing delay:
            transmit = 6.307e-03
```

```
            receive = 0.000e+00
          queueing delay = 3.144e-03
          net access delay = 2.300e-03
          mean wait delay = 5.444e-03

priority 8
   token cycle time = 1.524e-05
   mac queue length = 1.306e+00
   LLC layer queue length:
      transmit = 7.780e-01
       receive = 0.000e+00
   NETWORK layer queue length:
      transmit = 0.000e+00
       receive = 0.000e+00
   TRANSPORT layer queue length:
      transmit = 4.640e+01
       receive = 0.000e+00
   TRANSPORT layer queueing delay:
      transmit = 3.529e-03
       receive = 0.000e+00
   NETWORK layer queueing delay:
      transmit = 0.000e+00
       receive = 0.000e+00
   LLC layer queueing delay:
      transmit = 6.728e-03
       receive = 0.000e+00
   queueing delay = 3.111e-05
   net access delay = 2.116e-05
   mean wait delay = 5.227e-05
      class 0
         mac queue length = 1.278e+00
         queueing delay = 4.856e-06
         net access delay = 1.074e-05
         mean wait delay = 1.560e-05
      class 1
         mac queue length = 5.566e-02
         LLC layer queue length:
            transmit = 1.530e+00
             receive = 0.000e+00
         NETWORK layer queue length:
            transmit = 0.000e+00
             receive = 0.000e+00
         TRANSPORT layer queue length:
            transmit = 9.125e+01
             receive = 0.000e+00
         TRANSPORT layer queueing delay:
            transmit = 3.470e-03
             receive = 0.000e+00
         NETWORK layer queueing delay:
            transmit = 0.000e+00
             receive = 0.000e+00
```

```
LLC layer queueing delay:
   transmit = 6.669e-03
    receive = 0.000e+00
queueing delay = 3.253e-03
net access delay = 1.403e-03
mean wait delay = 4.656e-03
```

Most of the output is self-explanatory. The units used are *seconds* for delay, and *packets* for queue length. Note that if segmentation has taken place, the queue lengths are given in units of the reduced-length packets, not the original length packets. At the beginning of each output the date and time of the simulation run is given.

The notation

```
tra layer delay = 3.039e-03
rec tra layer delay = 0.000e+00
```

is interpreted as stating that the transport layer queueing delay when sending packets was 3.039e-03 seconds, and the transport layer queueing delay when receiving packets was 0.0 seconds. That is, any line beginning with `rec` indicates the measurement was taken at the receiving station. Any line not beginning with `rec` indicates the measurement was taken at the sending station.

Output is given first as an overall weighted average. Then it is broken down first by priority, and withing each priority level it is broken down by class. That is, if priority level 6 was included in both Class 2 and Class 7, then class 2 and class 7 data would be given under priority level 6. Output could have easily been sorted first by class and then by priority level; the method used here was arbitrarily chosen.

Mapping Actual Configurations to Simulation Language Code

It is a relatively simple matter to map an existing network configuration and message traffic distributions into the simulator programming language described above. Most of the commands of the language are self explanatory, e.g., the `circumference X` command, which allows the user to specify the total ring circumference. This can be further modified using the `meters N X` command to specify individual distances between stations. Using the `class N` commands, each station can be tailored to contribute the desired message traffic to the network load.

One problem that is inherent to simulation systems, but that may not be obvious at first sight is the fact that large demands are made on the computer performing the simulation in terms of both time and memory. It may happen that a perfectly legal simulation configuration may run on one machine but not on another due to the fact that the second machine lacks sufficient memory. Unfortunately, there is no way to solve this problem — it is up to the user to see that the computer on which the simulation is run has sufficient memory to allow the simulation to be completed.

As an example, consider the following simulator programming language code fragment:

```
class 0
includes 0 3 7 [10-14]
enter llc
load 65.0
length 32
priority 0 100.0
messages deterministic
arrivals deterministic
#
class 1
includes 1 2 4 5 6 8 9
enter tra
load 35.0
length 128
priority 0 100.0
messages exponential
arrivals poisson
```

This code fragment configures a network with two classes of messages. The first class enters at the LLC layer, contributes 0.65 of the total offered load of the simulation, has only a single priority level, and has deterministic message lengths and message arrival times. The second class enters at the transport layer, contributes 0.35 of the total offered load of the simulation, has a single priority level, and has exponentially distributed message lengths and a poisson arrival process for messages.

It would be a simple matter to add another message class such that at least some of the network stations offered more than a single message class to the network by the following code:

```
class 2
includes even 7
enter llc
load 15.0
length 64
priority 0 100.0
messages deterministic
arrivals poisson
```

Note that the `load X` command on at least one of the two previous message classes would have to be adjusted so that the total class load totals 100% of the total offered load of the simulation. Also not that if the code for class 2 had indicated two priority levels, class 2 would actually be two message classes — one for each priority level. This method would be a shorthand way of specifying two message classes which differed only in their priority level.

To control the simulation of the upper layers, we might have the following code fragment:

#

```
#
# process times for layers
# independent of class -- hardware/software dependent
# mac layer processing is computed by the simulation
#
# send and receive times are: ms to call, ns per byte in that order
# framing is in bytes
transport
  send 1.0 150.0
  rec 1.0 100.0
  framing 30
network
  send 0.0 0.0
  rec 0.0 0.0
  framing 0
llc
  send 0.0 1000.0
  rec 0.0 100.0
  framing 6
```

This fragment configures the upper layer call time and processing time, as well as giving the amount of framing each layer contributes to message length. In the above configuration, the network layer has been rendered inactive by setting all of its parameters to zero.

It should be noted that this fragment need not apply only to a network which uses the OSI protocols. The network to be simulated may be one that only has three levels of protocol operation. In this case the "transport" layer would simply be interpreted as the highest network processing layer, and the "LLC" layer would be the lowest. Thus layers need not be mapped to the corresponding OSI layers, but to any layer the user wishes merely by "reassigning" the names in the simulator programming language to the needed network layers.

Booting the Simulator

To compile the simulator, perform the following steps:

1) load the files into a single directory
2) type `make`
3) type `lex analyze.lex`
4) type `cc -o analyze lex.yy.c -ll`

The simulator is now compiled. It is not necessary to perform the above four steps again unless the simulator code has been updated.

The configuration to be simulated is compiled by performing the following steps:

1) edit the desired configuration file
2) type `parse <configuration file>`

The parsed configuration will be the one simulated. It is necessary to re-parse the configuration file each time the simulator is run.

To run the simulator using the desired configuration, type

```
simulate <output file>
```

This will send the results of the simulation to the file named `<output file>`. More information on running the simulator can be found in the language description.